

COP 4020: Programming Languages

Expressions and Assignment Statements

What is an expression?

Previous (unchanged)

```
(define-type Exp
  [numE (n : Number)]
  [plusE (left : Exp)
         (right : Exp)]
  [varE (name : Symbol)]
  [let1E (var : Symbol)
        (val : Exp)
        (body : Exp)]
  [lamE (var : Symbol)
        (body : Exp)]
  [appE (fun : Exp)
        (arg : Exp)]
)
```

New constructors added ◀

```
    ; --- core conditional ---
    [ifE (cnd : Exp) ;
condition
        (thn : Exp) ;
then-branch
        (els : Exp)] ;
else-branch

    ; --- syntactic sugar ---
    [andE (left : Exp) ; {and
e1 e2}
        (right : Exp)]

    [orE (left : Exp) ; {or
e1 e2}
        (right : Exp)]
```

What is an expression?

Everything is an expression:

- Arithmetic
- Conditionals
- Local bindings
- Variables
- Etc...

In Racket (the language to build languages): `(+ 3 (* 4 5))`

There is no ambiguity

Always clear the evaluation order designated by parentheses.

`“This is a / / / / /”`

Expressions and & Assignments

So far, this expression produces a value. At least this happens in the perfect world of languages we have built so far. This expression corresponds to the **surface syntax (programmer syntax)**,

expr --[]pipeline-[]value

Examples:

- $3+4$ or `{+ 3 4}`
- `if x>0 then x else -x` or `{if {> x 0} x {- x}}`
- `f(x) = x + 1` ; or `{lam {x} {+ x 1}}`
- `X=3` or `{let{x 3}} x`

The Two Worlds (Surface and AST)

Surface Syntax (what programmers write)

Examples from the slides:

$3 + 4 * 5$

$x = x + 1$

$(a \geq 0) \ \&\& \ (b < 10)$

Human-friendly

Ambiguous without rules

Requires:

- precedence
- associativity
- evaluation rules

AST (Abstract Syntax Tree)

Example: `let1E 'f(lamE 'x (plusE (varE 'x) (varE 'x))) (appE (varE 'f) (numE 3)))`

This is:

- Unambiguous
- Structured
- What the interpreter actually evaluates.

Precedence and associativity exist ONLY because we use surface syntax.

In AST form:

- You don't need precedence
- You don't need associativity
- The structure is already explicit

In this lecture, we are focusing on why real languages are messy

Surface syntax → Parsing → AST → Evaluation



```
graph LR; A((Surface syntax)) --> B[Parsing]; B --> C[AST]; C --> D[Evaluation]
```

This time, our focus will be on imperative languages

Arithmetic expressions

- In programming languages, arithmetic expressions consist of operators, operands, parentheses, and function calls.

Arithmetic expressions are built from:

Operands → values or variables {x, 5}

Operators → actions {+, *, -}

Parentheses → control evaluation order

Function calls → compute values {f(x)}

Expression: $3 + f(x) * (y - 2)$ or $\{+ 3 \{ * \{ f\ x \} \{- y\ 2 \} \} \}$
Why not lam?

Operands: 3, x, y, 2

Operators: +, *, -

Parentheses: (y - 2)

Function call: f(x)

Arithmetic expressions

- In programming languages, arithmetic expressions consist of operators, operands, parentheses, and function calls.

Arithmetic expressions are built from:

Operands → values or variables {x, 5}

Operators → actions {+, *, -}

Parentheses → control evaluation order

Function calls → compute values {f(x)}

Expression: $3 + f(x) * (y - 2)$ or $\{+ 3 \{ * \{ f x \} \{- y 2 \} \} \}$

Why not lam? Because we are using the function f(x), without defining the function

Operands: 3, x, y, 2

Operators: +, *, -

Parentheses: (y - 2)

Function call: f(x)

Arithmetic expressions

- In programming languages, arithmetic expressions consist of operators, operands, parentheses, and function calls.
- An operator can be unary, meaning it has a single operand, binary, meaning it has two operands, or ternary, meaning it has three operands.
- Infix operators appear between their operands, while prefix operators precede their operands.

a + b vs (+ a b)

Example: unary, binary, and ternary operators

Type	Operands	Example (C/Java)	S-expression
Unary	1	<code>-x</code>	<code>{- x}</code>
Binary	2	<code>a + b</code>	<code>{+ a b}</code>
Ternary	3	<code>c ? x : y</code>	<code>{if c x y}</code>

Design issues when designing an expression

Precedence

Associativity

Evaluation order

Side effects

Type mixing

Prefix vs Infix (Trade-off: clarity vs readability)

PLAI: (+ a b)

C/Java: a + b

(+ a (* b c))

Structure is explicit:

- No ambiguity
- No precedence rules needed

C/Java: a + b * c

Looks natural (like math), but:

- needs precedence rules
- can be confusing

The Problem: Ambiguity

$$3 + 4 * 5$$

- $(3+4)*5 = 35$
- $3+(4*5) = 23$

Which one?



Operator Precedence

* has higher precedence than +

$$3 + 4 * 5 = 23$$

Associativity

$$10 - 5 - 3 = (10-5)-3 \text{ or } 10 - (5 - 3)$$

$$2 ** 3 ** 2 = 2 ** (3 ** 2) \text{ or } (2 ** 3) ** 2$$



Common Precedence Hierarchy (C / Java style)

1. Unary operators

$-x$, $!x$, $++x$

2. Multiplicative

$*$, $/$, $\%$

3. Additive

$10 \&\& 5 < 6$

$+$, $-$

Evaluate: $2 + 3 * 4 >$

4. Relational

$<$, $>$, $<=$, $>=$

5. Equality

$==$, $!=$

6. Logical AND

$\&\&$

7. Logical OR

$\|\|$

8. Assignment (lowest)

$=$

Common Precedence Hierarchy (C / Java style)

Exercise: $2 + 3 * 4 > 10 \&\& 5 < 6$

Rule:

Unary, *,/,%, +,-, <,>,<=,>=, ==,!=, &&, ||, =

Evaluation order:

$3 * 4 = 12$

$2 + 12 = 14$

$14 > 10 = \text{true}$

$5 < 6 = \text{true}$

$\text{true} \&\& \text{true} = \text{true}$

What happens with same-precedence operators?

We use **associativity** to decide the order.

1. Left Associative (most common, in C) **left → right**

$$a + b + c \rightarrow (a + b) + c$$

$$a * b * c \rightarrow (a * b) * c$$

2. Right Associative Evaluate **right → left**

$$a ** b ** c \rightarrow a ** (b ** c)$$

$$a = b = c \rightarrow a = (b = c)$$

Another example

In C

Evaluate the following expressions

- a / b ; - $4/2 =$
- $a * b$; - $4*2 =$
- $a ** b$; - $4**2 =$

Another example

In C

- $a / b = (-a) / b$ unary - has **higher precedence**, but $-(a / b)$ produces the same result
- $a * b = (-a) * b$ unary - has **higher precedence**, but $-(a * b)$ produces the same result
- $a ** b$
- $a ** b \rightarrow (-a) ** b$ different from $-(a ** b)$

Another example

Unary minus has higher precedence than *, /, and **

- $a / b \rightarrow (-a) / b$ (same as $-(a / b)$)
- $a * b \rightarrow (-a) * b$ (same as $-(a * b)$)
- $a ** b \rightarrow (-a) ** b$ (different from $-(a ** b)$)

In C

`pow(-a, b)` // equivalent to $(-a)^b$

`-pow(a, b)` // equivalent to $-(a^b)$

In Python

- $a ** b \rightarrow -(a ** b)$

To change meaning:

$(-a) ** b \rightarrow$ different result

$$-2 ** 2 = -4$$

$$(-2) ** 2 = 4$$

Conclusion

Expressions seem simple... but:

- Syntax introduces ambiguity

- Languages add rules to fix it

Evaluation order

```
int a = 10;
```

```
int fun(int x) {  
    a = 20;  
    return 5;  
}
```

```
int b = a + fun(a);
```

Choices:

- A) 15
- B) 25
- C) 30
- D) Undefined / depends on evaluation order

Evaluation order (case 1: a evaluated first)

```
int a = 10;
```

```
int fun(int x) {  
    a = 20;  
    return 5;  
}
```

```
int b = a + fun(a);
```

Output

Read a = 10

Call fun(a):

sets a = 20

returns 5

Compute:

$10 + 5 = 15$

Evaluation order (case 2: f(a) evaluated first)

```
int a = 10;
```

```
int fun(int x) {  
    a = 20;  
    return 5;  
}
```

```
int b = a + fun(a);
```

Output

Call fun(a):

sets a = 20

returns 5

Read a → now 20

Compute:

$20 + 5 = 25$

Evaluation order (case 2: f(a) evaluated first)

```
int a = 10;
```

```
int fun(int x) {  
    a = 20;  
    return 5;  
}
```

```
int temp = fun(a);  
int b = a + temp;
```

Output

Call fun(a):
 sets a = 20
 returns 5

Read a → now 20

Compute:
 $20 + 5 = 25$

Design issues for arithmetic expressions

- What are the operator precedence rules?
- What are the operator associativity rules?
- What is the order of operand evaluation?
- Are there restrictions on operand evaluation side effects?
- Does the language allow user-defined operator overloading?
- What type mixing is allowed in expressions?

Operator Overloading

using the same operator symbol for different operations

Acceptable if:

- Readability is preserved
- Reliability is not affected

Operator Overloading (Example)

In C++

`a = &b;`

- Get the **memory address** of b
- Store it in a

`5 & 3`

Perform **bitwise AND** on a and b

// `101 & 011 = 001 → 1`

Another example (python)

- Operator overloading
 - Example: + operator
 - int + int: 3 + 4 # → 7
 - float + float; 3.5 + 2.1
 - array + array (what does it mean?): [1, 2] + [3, 4] # → [1, 2, 3, 4]
 - Strings: "Hello" + " World" # → "Hello World"

However:

```
import numpy as np
np.array([1,2]) + np.array([3,4]) # → [4 6]
```

Short-circuit evaluation

- A short-circuit evaluation of an expression is one in which the result is determined **without evaluating all of the operands and/or operators**
- **If the final result is already known, the rest of the expression is skipped.**

Example 1: if $a=0$

$(13 * a) * (b / 13 - 1) \neq 0 * (\text{anything}) = 0$
 $(b / 13 - 1)$ is not evaluated

Example 2: if $(a < 0)$

$(a \geq 0) \ \&\& \ (b < 10) \neq \text{false} \ \&\& \ \text{anything} = \text{false}$

An example of a potential problem with non-short-circuit evaluation of Boolean expressions

```
index = 0  
while ((index < listlen) && (list[index] != key))  
    index = index + 1
```

- In Java, the expression `(index < listlen) && (list[index] != key)` relies on short-circuit evaluation to prevent an index-out-of-bounds error.
- If Java didn't use short-circuit evaluation, both conditions `(index < listlen)` and `(list[index] != key)` would be evaluated regardless of the outcome of the first condition.
 - This could lead to an index-out-of-bounds error if `index` exceeds the bounds of the list array.

Danger

if expressions have **side effects**:

`(a > b) || ((b++) / 3)`

If the first part is true

`true || anything` $\square$ `true`

The second part is NOT executed, $\square$ `b++` does NOT happen

Summary

Short-circuit evaluation:

- Stops evaluating once result is known

Logical AND (&&):

`false && X → skip X`

Logical OR (||):

`true || X → skip X`

Benefits:

- efficiency
- safety

Danger:

- skipped side effects

Quiz question

```
int x = 0;
```

```
if ((x != 0) && (10 / x > 1)) {  
    printf("Yes");  
}
```

- A) Prints "Yes"
- B) Runtime error
- C) Nothing happens
- D) Undefined

Type conversions

A **narrowing conversion**: Converts a value to a smaller / less precise type

The new type cannot represent all values of the original type

```
double x = 3.14159;
```

```
float y = x; // narrowing
```

Type	Size	Precision
float	32 bits	~6–7 decimal digits
double	64 bits	~15–16 decimal digits

A **widening conversion**: Converts a value to a larger / more general type

```
int x = 5;
```

```
float y = x; // widening, no loss of info
```

- Explicit type conversions are called **casts**

```
int x = (int)3.7; // cast → 3
```

Relational expressions

- A relational operator is an operator that compares the values of its two operands.
- The value of a relational expression is Boolean
- JavaScript and PHP have two additional relational operators, `===` and `!==`. These are similar to their relatives, `==` and `!=`, but prevent their operands from being coerced.
 - For example, the expression
`"7" == 7`
is true in JavaScript, because when a string and a number are the operands of a relational operator, the string is coerced to a number. However,
`"7" === 7`
is false, because no coercion is done on the operands of this operator.

Quiz question

In java script

`"10" == 10`

- A) true
- B) false
- C) error
- D) undefined

Quiz question

In java script

`"10" === 10`

- A) true
- B) false
- C) error
- D) undefined

Chained comparison operators: C

Example

- One odd result of C's design of relational expressions is that the following expression is legal:

$a > b > c$

- Relational operators are **left-associative**
 $(a > b) > c$

Then

$a > b$ $\square$ this could be 1 (true) or 0 (false)

Now compare that result with c

$(0 \text{ or } 1) > c$

Example

$a = 5, b = 3, c = 2 ; a > b > c$

$5 > 3 > 2 \rightarrow (5 > 3) > 2 \rightarrow \text{true} (1) > 2 \rightarrow 1 > 2 \rightarrow 0 \text{ (false)}$

The problem is that we never compare b and c

Instead

$a > b > c$

It should be

$a > b \ \&\& \ b > c$

$a > b > c \rightarrow (a > b) > c$

Quiz question

```
int a = 5, b = 3, c = 2;  
printf("%d", a > b > c);
```

- A) 1
- B) 0
- C) true
- D) error

Compound and unary assignment operators

- Compound assignment operators

- Example

`a += b;` is equivalent to `a = a + b;`
`a -= b; // a = a - b`
`a *= b; // a = a * b`
`a /= b; // a = a / b`
`a %= b; // a = a % b`

- Unary assignment operators

- Example

`a = ++b;` is equivalent to `b = b + 1; a = b;`
`a = b++;` is equivalent to `a = b; b = b + 1;`

Compound and unary assignment operators

- Compound assignment operators

- Example

`a += b;` is equivalent to `a = a + b;`

- Unary assignment operators

- Example

`a = ++b;` is equivalent to `b = b + 1; a = b;`
`a = b++;` is equivalent to `a = b; b = b + 1;`
`++b`

Equivalent to:

```
b = b + 1;
return b;
```

```
int b = 5;
int a = ++b;
```

Step-by-step:

`b = 6` //after the increment

`a = 6` //return b after the

Compound and unary assignment operators

`b++`

Equivalent to:

`return b;`

`b = b + 1;`

`int b = 5;`

`int a = b++;`

Step-by-step:

`a = 5` //here is the return b before the increment

`b = 6` // here is b already incremented

Assignment as an expression

- **Example 1** Without the parentheses, the new character would be compared with EOF first. Then, the result of that comparison, either 0 or 1, would be assigned to ch.

```
while ((ch = getchar()) != EOF) {  
    ...  
}
```

```
while (ch = getchar() != EOF) {  
    ...  
}
```

- **Example 2**

a = b + (c = d / b) - 1

denotes the instructions

Assign d / b to c

Assign b + c to temp

Assign temp - 1 to a

Example 3

- There is a loss of error detection in the C design of the assignment operation that frequently leads to program errors. In particular, if we type

`if (x = y) ...`

instead of

`if (x == y) ...`

Multiple assignments

- Examples: PHP or Perl

```
($first, $second, $third) = (20, 40, 60);
```

The semantics is that 20 is assigned to \$first, 40 is assigned to \$second, and 60 is assigned to \$third.

In parallel

```
$first = 20
```

```
$second = 40
```

```
$third = 60
```

Swap

```
($first, $second) = ($second, $first);
```

```
$first = old $second
```

```
$second = old $first
```

This correctly interchanges the values of \$first and \$second, without the use of a temporary variable

Multiple assignments

In C

```
a = b;  
b = a;
```

Result:

Both become b

Right way in C

```
temp = a;  
a = b;  
b = temp
```

Language	Assignment style
PHP / Python	parallel assignment
C / Java	sequential assignment

Quiz question

In python

```
$a = 10;  
$b = 20;
```

```
($a, $b) = ($b, $a);
```

- A) a=10, b=20
- B) a=20, b=10
- C) a=20, b=20
- D) error

Evaluate right side first, then assign all at once